OpenAD Tutorial 1/2

Jean Utke[1]

[1]University of Chicago and Argonne National Laboratory

CMG Workshop
Sept. 9/10, 2009

outline

- part 1
  - motivation
  - AD basics and examples
  - reversal schemes and checkpointing
  - concerns for the AD user (model developer)
- part 2
  - non smooth models
  - checkpointing with Revolve
  - adjoinable MPI

why automatic differentiation?

given: some numerical model $\boldsymbol{y} = \boldsymbol{f}(\boldsymbol{x}) : \mathbb{R}^n \mapsto \mathbb{R}^m$ implemented as a (large / volatile) program

wanted: sensitivity analysis, optimization, parameter (state) estimation, higher-order approximation...

1. don't pretend we know nothing about the program
(and take finite differences of an oracle)
2. get machine precision derivatives as $\boldsymbol{J}\dot{\boldsymbol{x}}$ or $\bar{\boldsymbol{y}}^T\boldsymbol{J}$ or ...
(avoid approximation-versus-roundoff problem)
3. the reverse (aka adjoint) mode yields "cheap" gradients
4. if the program is large, so is the adjoint program, and
so is the effort to do it manually ... easy to get wrong but hard to debug

$\Rightarrow$ use tools to do it **automatically!**

why automatic differentiation?

given: some numerical model $\boldsymbol{y} = \boldsymbol{f}(\boldsymbol{x}) : \mathbb{R}^n \mapsto \mathbb{R}^m$ implemented as a (large / volatile) program

wanted: sensitivity analysis, optimization, parameter (state) estimation, higher-order approximation...

1. don't pretend we know nothing about the program
   (and take finite differences of an oracle)
2. get machine precision derivatives as $\boldsymbol{J}\dot{\boldsymbol{x}}$ or $\bar{\boldsymbol{y}}^T\boldsymbol{J}$ or ...
   (avoid approximation-versus-roundoff problem)
3. the reverse (aka adjoint) mode yields "cheap" gradients
4. if the program is large, so is the adjoint program, and
   so is the effort to do it manually ... easy to get wrong but hard to debug

$\Rightarrow$ use tools to do it **automatically?**

why automatic differentiation?

given: some numerical model $\boldsymbol{y} = \boldsymbol{f}(\boldsymbol{x}) : \mathbb{R}^n \mapsto \mathbb{R}^m$ implemented as a (large / volatile) program

wanted: sensitivity analysis, optimization, parameter (state) estimation, higher-order approximation...

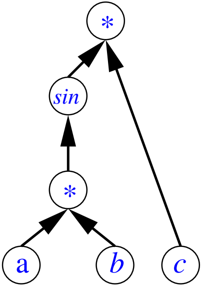
1. don't pretend we know nothing about the program
   (and take finite differences of an oracle)
2. get machine precision derivatives as $\boldsymbol{J}\dot{\boldsymbol{x}}$ or $\bar{\boldsymbol{y}}^T\boldsymbol{J}$ or ...
   (avoid approximation-versus-roundoff problem)
3. the reverse (aka adjoint) mode yields "cheap" gradients
4. if the program is large, so is the adjoint program, and
   so is the effort to do it manually ... easy to get wrong but hard to debug

$\Rightarrow$ use tools to do it at least semi-automatically! ☺

how does AD compute derivatives?

$f : y = sin(a * b) * c : \mathbb{R}^3 \mapsto \mathbb{R}$

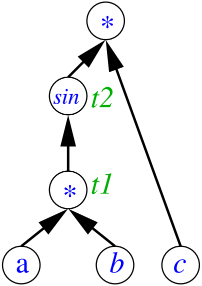
yields a graph representing the order of computation:



how does AD compute derivatives?

$f : y = sin(a * b) * c : \mathbb{R}^3 \mapsto \mathbb{R}$

yields a graph representing the order of computation:



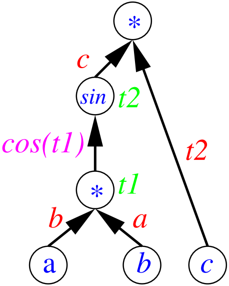
- *code list*→ intermediate values $t1$ and $t2$

```
t1 = a*b

t2 = sin(t1)
 y = t2*c
```

how does AD compute derivatives?

$f : y = sin(a * b) * c : \mathbb{R}^3 \mapsto \mathbb{R}$
yields a graph representing the order of computation:



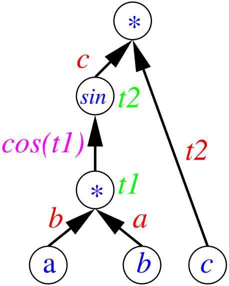
- *code list* $\rightarrow$ intermediate values $t1$ and $t2$
- each intrinsic $v = \phi(w, u)$ has local partials $\frac{\partial \phi}{\partial w}$, $\frac{\partial \phi}{\partial u}$
- e.g. `sin(t1)` yields `p1=cos(t1)`
- in our example all others are already stored in variables

```
t1 = a*b
p1 = cos(t1)
t2 = sin(t1)
 y = t2*c
```

how does AD compute derivatives?

$f : y = sin(a * b) * c : \mathbb{R}^3 \mapsto \mathbb{R}$

yields a graph representing the order of computation:



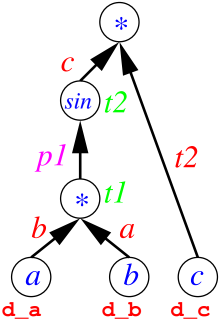
- *code list*$\rightarrow$ intermediate values $t1$ and $t2$
- each intrinsic $v = \phi(w, u)$ has local partials $\frac{\partial \phi}{\partial w}$, $\frac{\partial \phi}{\partial u}$
- e.g. `sin(t1)` yields `p1=cos(t1)`
- in our example all others are already stored in variables

```
t1 = a*b
p1 = cos(t1)
t2 = sin(t1)
 y = t2*c
```

What do we do with this?

forward mode with directional derivatives

- **associate** each variable v with a derivative $\dot{v}$
- take a point (a_0, b_0, c_0) and a direction $(\dot{a}, \dot{b}, \dot{c})$
- for each $v = \phi(w, u)$ propagate forward in order $\dot{v} = \frac{\partial\phi}{\partial w}\ \dot{w} + \frac{\partial\phi}{\partial u}\ \dot{u}$



- in practice: associate *by name* `[a,d_a]` or *by address* `[a%v,a%d]`
- interleave propagation computations

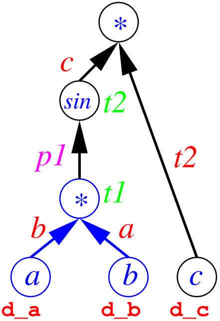
```
t1 = a*b

p1 = cos(t1)
t2 = sin(t1)

y = t2*c
```

forward mode with directional derivatives

- **associate** each variable v with a derivative $\dot{v}$
- take a point (a_0, b_0, c_0) and a direction $(\dot{a}, \dot{b}, \dot{c})$
- for each $v = \phi(w, u)$ propagate forward in order $\dot{v} = \frac{\partial \phi}{\partial w}\ \dot{w} + \frac{\partial \phi}{\partial u}\ \dot{u}$



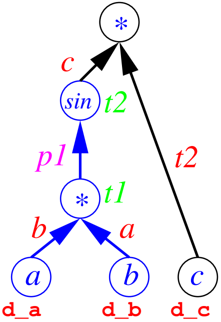
- in practice: associate *by name* `[a,d_a]` or *by address* `[a%v,a%d]`
- interleave propagation computations

```
t1  = a*b
d_t1 = d_a*b + d_b*a
p1  = cos(t1)
t2  = sin(t1)

y  = t2*c
```

forward mode with directional derivatives

- **associate** each variable v with a derivative $\dot{v}$
- take a point (a_0, b_0, c_0) and a direction $(\dot{a}, \dot{b}, \dot{c})$
- for each $v = \phi(w, u)$ propagate forward in order $\dot{v} = \frac{\partial \phi}{\partial w} \dot{w} + \frac{\partial \phi}{\partial u} \dot{u}$

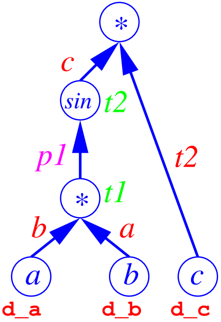


- in practice: associate *by name* `[a,d_a]`
  or *by address* `[a%v,a%d]`
- interleave propagation computations

```
t1 = a*b
d_t1 = d_a*b + d_b*a
p1 = cos(t1)
t2 = sin(t1)
d_t2 = d_t1*p1
y = t2*c
```

forward mode with directional derivatives

- **associate** each variable v with a derivative $\dot{v}$
- take a point (a_0, b_0, c_0) and a direction $(\dot{a}, \dot{b}, \dot{c})$
- for each $v = \phi(w, u)$ propagate forward in order $\dot{v} = \frac{\partial\phi}{\partial w} \; \dot{w} \; + \; \frac{\partial\phi}{\partial u} \; \dot{u}$

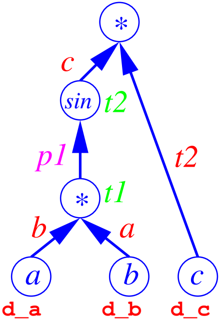


- in practice: associate *by name* `[a,d_a]`
  or *by address* `[a%v,a%d]`
- interleave propagation computations

```
t1  = a*b
d_t1 = d_a*b + d_b*a
p1  = cos(t1)
t2  = sin(t1)
d_t2 = d_t1*p1
y  = t2*c
d_y = d_t2*c + d_c*t2
```

forward mode with directional derivatives

- **associate** each variable v with a derivative $\dot{v}$
- take a point (a_0, b_0, c_0) and a direction $(\dot{a}, \dot{b}, \dot{c})$
- for each $v = \phi(w, u)$ propagate forward in order $\dot{v} = \frac{\partial \phi}{\partial w} \; \dot{w} + \frac{\partial \phi}{\partial u} \; \dot{u}$



- in practice: associate *by name* `[a,d_a]` or *by address* `[a%v,a%d]`
- interleave propagation computations

```
t1 = a*b
d_t1 = d_a*b + d_b*a
p1 = cos(t1)
t2 = sin(t1)
d_t2 = d_t1*p1
y = t2*c
d_y = d_t2*c + d_c*t2
```

What is in **d_y** ?

`d_y` contains a projection

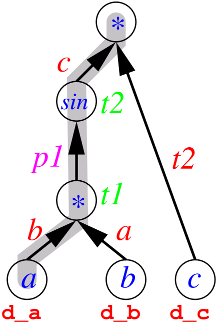
- $\dot{\boldsymbol{y}} = \boldsymbol{J}\dot{\boldsymbol{x}}$ computed at $\boldsymbol{x}_0$

`d_y` contains a projection

- $\dot{\boldsymbol{y}} = \boldsymbol{J}\dot{\boldsymbol{x}}$ computed at $\boldsymbol{x}_0$
- for example for $(\dot{a}, \dot{b}, \dot{c}) = (1, 0, 0)$

d_y contains a projection

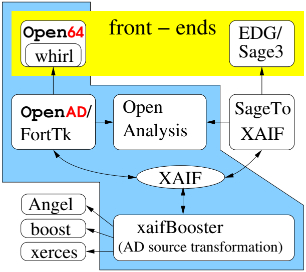
- $\dot{\boldsymbol{y}} = \boldsymbol{J}\dot{\boldsymbol{x}}$ computed at $\boldsymbol{x}_0$
- for example for $(\dot{a}, \dot{b}, \dot{c}) = (1, 0, 0)$



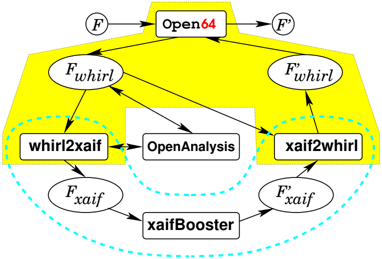
- yields the first element of the gradient
- all gradient elements cost $\mathcal{O}(n)$ function evaluations

sidebar: OpenAD overview

- www.mcs.anl.gov/OpenAD
- forward and **reverse**
- source transformation
- modular design
- aims at large problems
- language independent transformation
- researching combinatorial problems
- current Fortran front-end Open64 (Open64/SL branch at Rice U)
- migration to Rose (already used for C/C++ with EDG)
- uses *association by address* (i.e. has an active type)
- Rapsodia for higher-order derivatives via type change transformation



Fortran pipeline:



sidebar: toy example

numerical "model" program:

```
subroutine head(x,y)
  double precision,intent(in) :: x
  double precision,intent(out) :: y

  y=sin(x*x)

end subroutine
```

sidebar: toy example

prepare original code

numerical “model” program:

```
subroutine head(x,y)
  double precision,intent(in) :: x
  double precision,intent(out) :: y
!$openad INDEPENDENT(x)
  y=sin(x*x)
!$openad DEPENDENT(y)
end subroutine
```

sidebar: toy example

prepare original code ⇒ run it through OpenAD

numerical "model" program:

```
subroutine head(x,y)
  double precision,intent(in) :: x
  double precision,intent(out) :: y
!$openad INDEPENDENT(x)
  y=sin(x*x)
!$openad DEPENDENT(y)
end subroutine
```

transformed model program:

```
SUBROUTINE head(X, Y)
use w2f__types
use OAD_active
IMPLICIT NONE
REAL(w2f__8) oadS_0
...
REAL(w2f__8) oadS_5
type(active) :: X
INTENT(IN) X
type(active) :: Y
INTENT(OUT) Y
oadS_0 = (X%v*X%v)
Y%v = SIN(oadS_0)
oadS_2 = X%v
oadS_3 = X%v
oadS_1 = COS(oadS_0)
oadS_5 = ((oadS_3 + oadS_2) * oadS_1)
CALL sax(oadS_5,X,Y)
RETURN
END SUBROUTINE
```

sidebar: toy example

prepare original code ⇒ run it through OpenAD

numerical “model” program:

```
subroutine head(x,y)
  double precision,intent(in) :: x
  double precision,intent(out) :: y
!$openad INDEPENDENT(x)
  y=sin(x*x)
!$openad DEPENDENT(y)
end subroutine
```

transformed model program:

```
SUBROUTINE head(X, Y)
use w2f__types
use OAD_active
IMPLICIT NONE
REAL(w2f__8) oadS_0
...
REAL(w2f__8) oadS_5
type(active) :: X
INTENT(IN) X
type(active) :: Y
INTENT(OUT) Y
oadS_0 = (X%v*X%v)
Y%v = SIN(oadS_0)
oadS_2 = X%v
oadS_3 = X%v
oadS_1 = COS(oadS_0)
oadS_5 = ((oadS_3 + oadS_2) * oadS_1)
CALL sax(oadS_5,X,Y)
RETURN
END SUBROUTINE
```

sidebar: toy example

prepare original code ⇒ run it through OpenAD

numerical "model" program:

```
subroutine head(x,y)
  double precision,intent(in) :: x
  double precision,intent(out) :: y
!$openad INDEPENDENT(x)
  y=sin(x*x)
!$openad DEPENDENT(y)
end subroutine
```

transformed model program:

```
SUBROUTINE head(X, Y)
use w2f__types
use OAD_active
IMPLICIT NONE
REAL(w2f__8) oadS_0
...
REAL(w2f__8) oadS_5
type(active) :: X
INTENT(IN)  X
type(active) :: Y
INTENT(OUT)  Y
oadS_0 = (X%v*X%v)
Y%v = SIN(oadS_0)
oadS_2 = X%v
oadS_3 = X%v
oadS_1 = COS(oadS_0)
oadS_5 = ((oadS_3 + oadS_2) * oadS_1)
CALL sax(oadS_5,X,Y)
RETURN
END SUBROUTINE
```

sidebar: toy example

prepare original code ⇒ run it through OpenAD ⇒ adapt a driver routine

numerical "model" program:

```
subroutine head(x,y)
  double precision,intent(in) :: x
  double precision,intent(out) :: y
!$openad INDEPENDENT(x)
  y=sin(x*x)
!$openad DEPENDENT(y)
end subroutine
```

driver:

```
program driver

  implicit none
  external head
  real:: x, y
  x=.5D0

  call head(x,y)

end program driver
```

transformed model program:

```
SUBROUTINE head(X, Y)
use w2f__types
use OAD_active
IMPLICIT NONE
REAL(w2f__8) oadS_0
...
REAL(w2f__8) oadS_5
type(active) :: X
INTENT(IN)  X
type(active) :: Y
INTENT(OUT)  Y
oadS_0 = (X%v*X%v)
Y%v = SIN(oadS_0)
oadS_2 = X%v
oadS_3 = X%v
oadS_1 = COS(oadS_0)
oadS_5 = ((oadS_3 + oadS_2) * oadS_1)
CALL sax(oadS_5,X,Y)
RETURN
END SUBROUTINE
```

sidebar: toy example

prepare original code ⇒ run it through OpenAD ⇒ adapt a driver routine

numerical “model” program:

```
subroutine head(x,y)
  double precision,intent(in) :: x
  double precision,intent(out) :: y
!$openad INDEPENDENT(x)
  y=sin(x*x)
!$openad DEPENDENT(y)
end subroutine
```

driver:

```
program driver
  use OAD_active
  implicit none
  external head
  type(active):: x, y
  x%v=.5D0
  x%d=1.0
  call head(x,y)
  print *, "F(1,1)=",y%d
end program driver
```

transformed model program:

```
SUBROUTINE head(X, Y)
use w2f__types
use OAD_active
IMPLICIT NONE
REAL(w2f__8) oadS_0
...
REAL(w2f__8) oadS_5
type(active) :: X
INTENT(IN)  X
type(active) :: Y
INTENT(OUT)  Y
oadS_0 = (X%v*X%v)
Y%v = SIN(oadS_0)
oadS_2 = X%v
oadS_3 = X%v
oadS_1 = COS(oadS_0)
oadS_5 = ((oadS_3 + oadS_2) * oadS_1)
CALL sax(oadS_5,X,Y)
RETURN
END SUBROUTINE
```

sidebar: toy example

prepare original code ⇒ run it through OpenAD ⇒ adapt a driver routine

numerical "model" program:

```
subroutine head(x,y)
  double precision,intent(in) :: x
  double precision,intent(out) :: y
!$openad INDEPENDENT(x)
  y=sin(x*x)
!$openad DEPENDENT(y)
end subroutine
```

driver:

```
program driver
  use OAD_active
  implicit none
  external head
  type(active):: x, y
  x%v=.5D0
  x%d=1.0
  call head(x,y)
  print *, "F(1,1)=",y%d
end program driver
```

transformed model program:

```
SUBROUTINE head(X, Y)
use w2f__types
use OAD_active
IMPLICIT NONE
REAL(w2f__8) oadS_0
...
REAL(w2f__8) oadS_5
type(active) :: X
INTENT(IN)  X
type(active) :: Y
INTENT(OUT)  Y
oadS_0 = (X%v*X%v)
Y%v = SIN(oadS_0)
oadS_2 = X%v
oadS_3 = X%v
oadS_1 = COS(oadS_0)
oadS_5 = ((oadS_3 + oadS_2) * oadS_1)
CALL sax(oadS_5,X,Y)
RETURN
END SUBROUTINE
```

the `sax` call comes from propagation following *preaccumulation*...not discussed yet

try it out

- `cd ~/OpenAD`
- `.  ./setenv.sh`
- this sets up environment variables (see below) and some shell aliases

```
ANGELROOT=/home/guest1/OpenAD/angel
ANGEL_BASE=/home/guest1/OpenAD/angel
BOOSTROOT=/home/guest1/OpenAD/boost
BOOST_BASE=/home/guest1/OpenAD/boost
LD_LIBRARY_PATH=/home/guest1/OpenAD/Open64/osprey1.0/targ_ia32_ia64_linux/whirl2f:\
  /opt/intel/Compiler/11.0/083/lib/ia32
OPEN64ROOT=/home/guest1/OpenAD/Open64/osprey1.0/targ_ia32_ia64_linux
OPEN64TARG=targ_ia32_ia64_linux
OPEN64_BASE=/home/guest1/OpenAD/Open64
OPENADFORTTK=/home/guest1/OpenAD/OpenADFortTk/OpenADFortTk-x86-Linux
OPENADFORTTKROOT=/home/guest1/OpenAD/OpenADFortTk/OpenADFortTk-x86-Linux
OPENADFORTTK_BASE=/home/guest1/OpenAD/OpenADFortTk
OPENADPLATFORM=x86-Linux
OPENADROOT=/home/guest1/OpenAD
OPENAD_BASE=/home/guest1/OpenAD
OPENANALYSISROOT=/home/guest1/OpenAD/OpenAnalysis/x86-Linux
OPENANALYSIS_BASE=/home/guest1/OpenAD/OpenAnalysis
PATH=/home/guest1/OpenAD/bin:/home/guest1/OpenAD/OpenADFortTk/OpenADFortTk-x86-Linuxbin:...
REVOLVEF9XROOT=/home/guest1/OpenAD/RevolveF9X
XAIFBOOSTERROOT=/home/guest1/OpenAD/xaifBooster/..
XAIFBOOSTER_BASE=/home/guest1/OpenAD/xaifBooster
XAIFSCHEMAROOT=/home/guest1/OpenAD/xaif
XAIFSCHEMA_BASE=/home/guest1/OpenAD/xaif
XERCESCROOT=/home/guest1/OpenAD/xercesc/x86-Linux
XERCESC_BASE=/home/guest1/OpenAD/xercesc
```

try it out contd. ...

- `cd Examples/OneMinute`
- look at `head.prepped.f90` vs. `head.f90`
- look at the `Makefile`
- `openad -h` to see some options
- run `make clean`
- and `make`

```
openad -c -m f head.prepped.f90
openad log: openad.2009-09-05_16:19:02.log~
parsing head.prepped.f90
analyzing source code and translating to xaif
tangent linear transformation
 getting runtime support file OAD_active.f90
   getting runtime support file w2f__types.f90
 getting runtime support file iaddr.c
translating transformed xaif to whirl
unparsing transformed whirl to fortran
postprocessing transformed fortran
gfortran -o w2f__types.o -c w2f__types.f90
gfortran -o OAD_active.o -c OAD_active.f90
gfortran -o driver.o -c driver.f90
gfortran -o head.prepped.xb.x2w.w2f.pp.o -c head.prepped.xb.x2w.w2f.pp.f
gfortran -o driver w2f__types.o OAD_active.o driver.o head.prepped.xb.x2w.w2f.pp.o
```

try it out contd. ...

- look at the transformed file `head.prepped.xb.x2w.w2f.pp.f`
- look at the the driver code in `driver.f90`
- run the binary: `./driver`
- should produce output like this:

```
/home/guest1/OpenAD/Examples/OneMinute> ./driver
 driver running for x =   0.50000000000000000
            yields y =   0.54630248984379048       dy/dx =    1.2984464104095248
     1+tan(x)^2-dy/dx =   3.85975973404839579E-017
```

applications

for instance

- ocean/atmosphere state estimation & uncertainty quantification, oil reservoir modeling
- computational chemical engineering
- airfoil shape optimization, suspended droplets, ...
- beam physics
- mechanical engineering (design optimization)

use

- **gradients**
- Jacobian projections
- Hessian projections
- higher order derivatives
  (full or partial tensors, univariate Taylor series)

applications

for instance

- ocean/atmosphere state estimation & uncertainty quantification, oil reservoir modeling
- computational chemical engineering
- airfoil shape optimization, suspended droplets, ...
- beam physics
- mechanical engineering (design optimization)

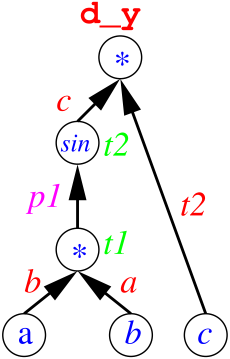
use

- **gradients**
- Jacobian projections
- Hessian projections
- higher order derivatives
  (full or partial tensors, univariate Taylor series)

How do we get the cheap gradients?

reverse mode with adjoints

- same association model
- take a point (a_0, b_0, c_0), compute y, pick a weight $\bar{y}$
- for each $v = \phi(w, u)$ propagate backward
  $\bar{w} += \frac{\partial \phi}{\partial w} \bar{v}; \quad \bar{u} += \frac{\partial \phi}{\partial u} \bar{v}; \quad \bar{v} = 0$

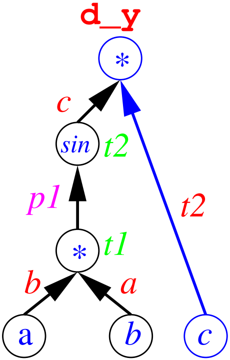


backward propagation code appended:

```
t1 = a*b
p1 = cos(t1)
t2 = sin(t1)
y = t2*c
```

reverse mode with adjoints

- same association model
- take a point (a_0, b_0, c_0), compute y, pick a weight $\bar{y}$
- for each $v = \phi(w, u)$ propagate backward
  $\bar{w} += \frac{\partial\phi}{\partial w}\ \bar{v}; \quad \bar{u} += \frac{\partial\phi}{\partial u}\ \bar{v}; \quad \bar{v} = 0$

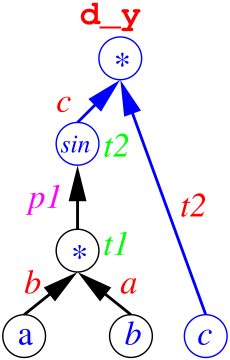


backward propagation code appended:

```
t1 = a*b
p1 = cos(t1)
t2 = sin(t1)
y = t2*c
d_c = t2*d_y
```

reverse mode with adjoints

- same association model
- take a point (a_0, b_0, c_0), compute y, pick a weight $\bar{y}$
- for each $v = \phi(w, u)$ propagate backward
  $\bar{w} += \frac{\partial \phi}{\partial w} \; \bar{v}; \quad \bar{u} += \frac{\partial \phi}{\partial u} \; \bar{v}; \quad \bar{v} = 0$

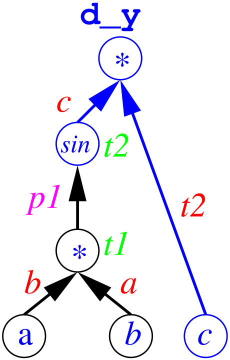


backward propagation code appended:

```
t1 = a*b
p1 = cos(t1)
t2 = sin(t1)
y = t2*c
d_c = t2*d_y
d_t2 = c*d_y
```

reverse mode with adjoints

- same association model
- take a point (a_0, b_0, c_0), compute y, pick a weight $\bar{y}$
- for each $v = \phi(w, u)$ propagate backward
  $\bar{w}+ = \frac{\partial \phi}{\partial w} \ \bar{v}; \quad \bar{u}+ = \frac{\partial \phi}{\partial u} \ \bar{v}; \quad \bar{v} = 0$

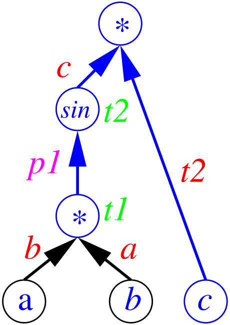


backward propagation code appended:

```
t1 = a*b
p1 = cos(t1)
t2 = sin(t1)
y = t2*c
d_c = t2*d_y
d_t2 = c*d_y
d_y = 0
```

reverse mode with adjoints

- same association model
- take a point (a_0, b_0, c_0), compute y, pick a weight $\bar{y}$
- for each $v = \phi(w, u)$ propagate backward
  $\bar{w}+ = \frac{\partial\phi}{\partial w}\ \bar{v};\quad \bar{u}+ = \frac{\partial\phi}{\partial u}\ \bar{v};\quad \bar{v} = 0$

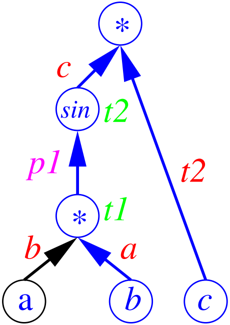


backward propagation code appended:

```
t1  = a*b
p1  = cos(t1)
t2  = sin(t1)
y  = t2*c
d_c  = t2*d_y
d_t2  = c*d_y
d_y  = 0
d_t1  = p1*d_t2
```

reverse mode with adjoints

- same association model
- take a point (a_0, b_0, c_0), compute y, pick a weight $\bar{y}$
- for each $v = \phi(w, u)$ propagate backward
  $\bar{w} += \frac{\partial \phi}{\partial w}\ \bar{v}; \quad \bar{u} += \frac{\partial \phi}{\partial u}\ \bar{v}; \quad \bar{v} = 0$

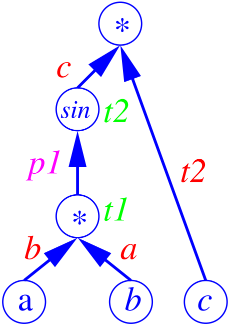


backward propagation code appended:

```
t1 = a*b
p1 = cos(t1)
t2 = sin(t1)
y = t2*c
d_c = t2*d_y
d_t2 = c*d_y
d_y = 0
d_t1 = p1*d_t2
d_b = a*d_t1
```

reverse mode with adjoints

- same association model
- take a point (a_0, b_0, c_0), compute y, pick a weight $\bar{y}$
- for each $v = \phi(w, u)$ propagate backward
  $\bar{w} += \frac{\partial \phi}{\partial w} \ \bar{v}; \quad \bar{u} += \frac{\partial \phi}{\partial u} \ \bar{v}; \quad \bar{v} = 0$

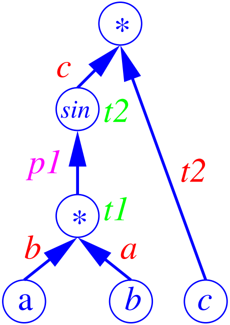


backward propagation code appended:

```
t1 = a*b
p1 = cos(t1)
t2 = sin(t1)
y = t2*c
d_c = t2*d_y
d_t2 = c*d_y
d_y = 0
d_t1 = p1*d_t2
d_b = a*d_t1
d_a = b*d_t1
```

reverse mode with adjoints

- same association model
- take a point (a_0, b_0, c_0), compute y, pick a weight $\bar{y}$
- for each $v = \phi(w, u)$ propagate backward
  $\bar{w}+ = \frac{\partial\phi}{\partial w}\ \bar{v}; \quad \bar{u}+ = \frac{\partial\phi}{\partial u}\ \bar{v}; \quad \bar{v} = 0$



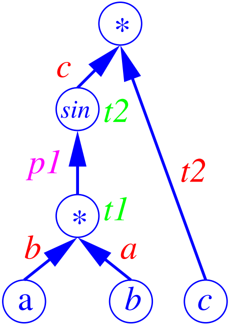
backward propagation code appended:

```
t1  = a*b
p1  = cos(t1)
t2  = sin(t1)
y  = t2*c
d_c  = t2*d_y
d_t2  = c*d_y
d_y  = 0
d_t1  = p1*d_t2
d_b  = a*d_t1
d_a  = b*d_t1
```

What is in (d_a, d_b, d_c)?

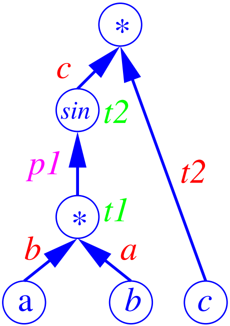
(d_a, d_b, d_c) contains a projection

- $\bar{\boldsymbol{x}} = \bar{\boldsymbol{y}}^T \boldsymbol{J}$ computed at $\boldsymbol{x}_0$



(d_a, d_b, d_c) contains a projection

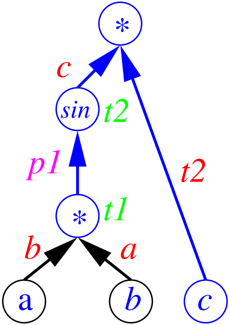
- $\bar{\boldsymbol{x}} = \bar{\boldsymbol{y}}^T \boldsymbol{J}$ computed at $\boldsymbol{x}_0$
- for example for $\bar{y} = 1$ we have $[\bar{a}, \bar{b}, \bar{c}] = \nabla f$



- all gradient elements cost $\mathcal{O}(1)$ function evaluations

(d_a, d_b, d_c) contains a projection

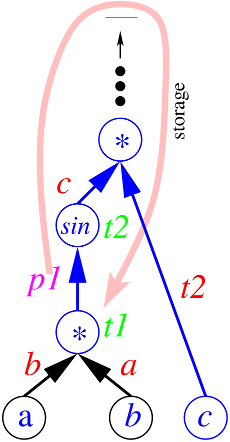
- $\bar{\boldsymbol{x}} = \bar{\boldsymbol{y}}^T \boldsymbol{J}$ computed at $\boldsymbol{x}_0$
- for example for $\bar{y} = 1$ we have $[\bar{a}, \bar{b}, \bar{c}] = \nabla f$



- all gradient elements cost $\mathcal{O}(1)$ function evaluations
- but consider when $p1$ is computed and when it is used

$(\texttt{d\_a}, \texttt{d\_b}, \texttt{d\_c})$ contains a projection

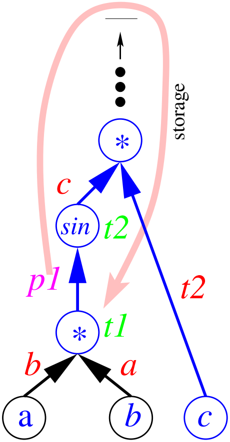
- $\bar{\boldsymbol{x}} = \bar{\boldsymbol{y}}^T \boldsymbol{J}$ computed at $\boldsymbol{x}_0$
- for example for $\bar{y} = 1$ we have $[\bar{a}, \bar{b}, \bar{c}] = \nabla f$



- all gradient elements cost $\mathcal{O}(1)$ function evaluations
- but consider when $p1$ is computed and when it is used
- storage requirements grow with the length of the computation
- typically mitigated by recomputation from checkpoints

(`d_a`, `d_b`, `d_c`) contains a projection

- $\bar{\boldsymbol{x}} = \bar{\boldsymbol{y}}^T \boldsymbol{J}$ computed at $\boldsymbol{x}_0$
- for example for $\bar{y} = 1$ we have $[\bar{a}, \bar{b}, \bar{c}] = \nabla f$

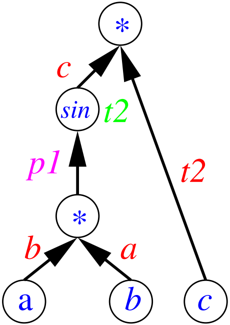


- all gradient elements cost $\mathcal{O}(1)$ function evaluations
- but consider when $p1$ is computed and when it is used
- storage requirements grow with the length of the computation
- typically mitigated by recomputation from checkpoints

Reverse mode as a source transformation with OpenAD.

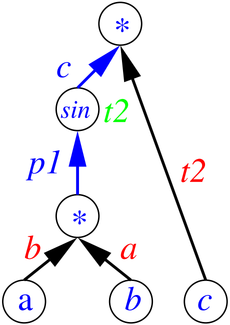
sidebar: preaccumulation & propagation

- build expression graphs (limited by aliasing, typically to a basic block)
- **preaccumulate** them to local Jacobians $\boldsymbol{J}$
- long program with control flow $\Rightarrow$ sequence of graphs $\Rightarrow$ sequence of $\boldsymbol{J}_i$



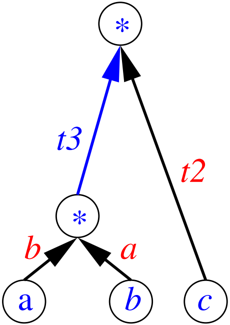
sidebar: preaccumulation & propagation

- build expression graphs (limited by aliasing, typically to a basic block)
- **preaccumulate** them to local Jacobians $\boldsymbol{J}$
- long program with control flow ⇒ sequence of graphs ⇒ sequence of $\boldsymbol{J}_i$



sidebar: preaccumulation & propagation

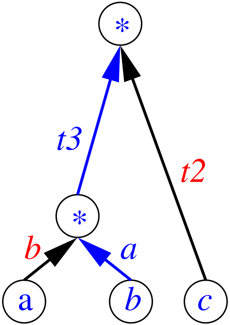
- build expression graphs (limited by aliasing, typically to a basic block)
- **preaccumulate** them to local Jacobians $\boldsymbol{J}$
- long program with control flow ⇒ sequence of graphs ⇒ sequence of $\boldsymbol{J}_i$




```
t3 = c*p1
```

sidebar: preaccumulation & propagation

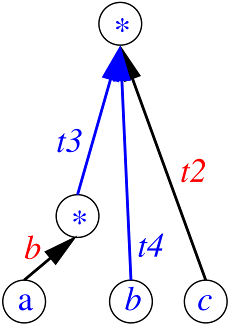
- build expression graphs (limited by aliasing, typically to a basic block)
- **preaccumulate** them to local Jacobians $\boldsymbol{J}$
- long program with control flow $\Rightarrow$ sequence of graphs $\Rightarrow$ sequence of $\boldsymbol{J}_i$




```
t3 = c*p1
```

sidebar: preaccumulation & propagation

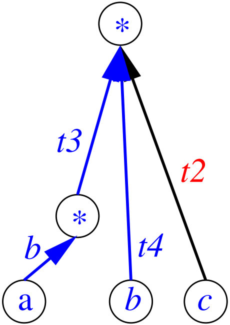
- build expression graphs (limited by aliasing, typically to a basic block)
- **preaccumulate** them to local Jacobians $\boldsymbol{J}$
- long program with control flow $\Rightarrow$ sequence of graphs $\Rightarrow$ sequence of $\boldsymbol{J}_i$




```
t3 = c*p1
t4 = t3*a
```

sidebar: preaccumulation & propagation

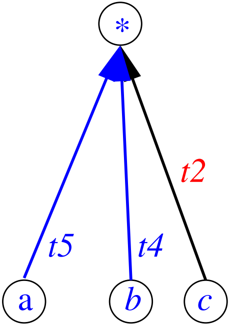
- build expression graphs (limited by aliasing, typically to a basic block)
- **preaccumulate** them to local Jacobians $\boldsymbol{J}$
- long program with control flow $\Rightarrow$ sequence of graphs $\Rightarrow$ sequence of $\boldsymbol{J}_i$




```
t3 = c*p1
t4 = t3*a
```

sidebar: preaccumulation & propagation

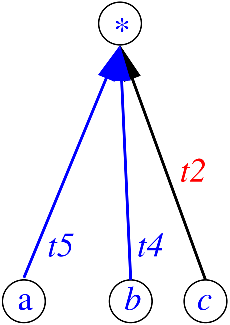
- build expression graphs (limited by aliasing, typically to a basic block)
- **preaccumulate** them to local Jacobians $\boldsymbol{J}$
- long program with control flow ⇒ sequence of graphs ⇒ sequence of $\boldsymbol{J}_i$




```
t3 = c*p1
t4 = t3*a
t5 = t3*b
```

sidebar: preaccumulation & propagation

- build expression graphs (limited by aliasing, typically to a basic block)
- **preaccumulate** them to local Jacobians $\boldsymbol{J}$
- long program with control flow $\Rightarrow$ sequence of graphs $\Rightarrow$ sequence of $\boldsymbol{J}_i$




```
t3 = c*p1
t4 = t3*a
t5 = t3*b
```

- (`t5,t4,t2`) is the preaccumulated $\boldsymbol{J}_i$
- $\min_{ops}$(preaccumulation) ?
  is a combinatorial problem
  $\Rightarrow$ compile time AD optimization!
- forward propagation of $\dot{\boldsymbol{x}}$
  $(\boldsymbol{J}_k \circ \ldots \circ (\boldsymbol{J}_1 \circ \dot{\boldsymbol{x}}) \ldots)$
- adjoint propagation of $\bar{\boldsymbol{y}}$
  $(\ldots(\bar{\boldsymbol{y}}^T \circ \boldsymbol{J}_k) \circ \ldots \circ \boldsymbol{J}_1)$

sidebar: toy example - reverse mode

same code preparation

numerical “model” program:

```
subroutine head(x,y)
  double precision,intent(in) :: x
  double precision,intent(out) :: y
!$openad INDEPENDENT(x)
  y=sin(x*x)
!$openad DEPENDENT(y)
end subroutine
```

sidebar: toy example - reverse mode

same code preparation ⇒ reverse mode OpenAD pipeline

numerical "model" program:

```
subroutine head(x,y)
  double precision,intent(in) :: x
  double precision,intent(out) :: y
!$openad INDEPENDENT(x)
  y=sin(x*x)
!$openad DEPENDENT(y)
end subroutine
```

preaccumulation & store $\boldsymbol{J}_i$:

```
...
oadS_0 = (X%v*X%v)
Y%v = SIN(oadS_0)
oadS_2 = X%v
oadS_3 = X%v
oadS_1 = COS(oadS_0)
oadS_4 = (oadS_2 * oadS_1)
oadS_5 = (oadS_3 * oadS_1)
oadD(oadD_ptr) = oadS_4
oadD_ptr = oadD_ptr+1
oadD(oadD_ptr) = oadS_5
oadD_ptr = oadD_ptr+1
...
```

sidebar: toy example - reverse mode

same code preparation $\Rightarrow$ reverse mode OpenAD pipeline

numerical "model" program:

```
subroutine head(x,y)
  double precision,intent(in) :: x
  double precision,intent(out) :: y
!$openad INDEPENDENT(x)
  y=sin(x*x)
!$openad DEPENDENT(y)
end subroutine
```

preaccumulation & store J_i:

```
...
oadS_0 = (X%v*X%v)
Y%v = SIN(oadS_0)
oadS_2 = X%v
oadS_3 = X%v
oadS_1 = COS(oadS_0)
oadS_4 = (oadS_2 * oadS_1)
oadS_5 = (oadS_3 * oadS_1)
oadD(oadD_ptr) = oadS_4
oadD_ptr = oadD_ptr+1
oadD(oadD_ptr) = oadS_5
oadD_ptr = oadD_ptr+1
...
```

sidebar: toy example - reverse mode

same code preparation ⇒ reverse mode OpenAD pipeline

numerical "model" program:

```
subroutine head(x,y)
  double precision,intent(in) :: x
  double precision,intent(out) :: y
!$openad INDEPENDENT(x)
  y=sin(x*x)
!$openad DEPENDENT(y)
end subroutine
```

preaccumulation & store J_i:

```
...
oadS_0 = (X%v*X%v)
Y%v = SIN(oadS_0)
oadS_2 = X%v
oadS_3 = X%v
oadS_1 = COS(oadS_0)
oadS_4 = (oadS_2 * oadS_1)
oadS_5 = (oadS_3 * oadS_1)
oadD(oadD_ptr) = oadS_4
oadD_ptr = oadD_ptr+1
oadD(oadD_ptr) = oadS_5
oadD_ptr = oadD_ptr+1
...
```

sidebar: toy example - reverse mode

same code preparation ⇒ reverse mode OpenAD pipeline

numerical "model" program:

```
subroutine head(x,y)
  double precision,intent(in) :: x
  double precision,intent(out) :: y
!$openad INDEPENDENT(x)
  y=sin(x*x)
!$openad DEPENDENT(y)
end subroutine
```

preaccumulation & store J_i:

```
...
oadS_0 = (X%v*X%v)
Y%v = SIN(oadS_0)
oadS_2 = X%v
oadS_3 = X%v
oadS_1 = COS(oadS_0)
oadS_4 = (oadS_2 * oadS_1)
oadS_5 = (oadS_3 * oadS_1)
oadD(oadD_ptr) = oadS_4
oadD_ptr = oadD_ptr+1
oadD(oadD_ptr) = oadS_5
oadD_ptr = oadD_ptr+1
...
```

retrieve stored J_i & propagate:

```
...
oadD_ptr = oadD_ptr-1
oadS_6 = oadD(oadD_ptr)
X%d = X%d+Y%d*oadS_6
oadD_ptr = oadD_ptr-1
oadS_7 = oadD(oadD_ptr)
X%d = X%d+Y%d*oadS_7
Y%d = 0.0d0
...
```

sidebar: toy example - reverse mode

same code preparation ⇒ reverse mode OpenAD pipeline

numerical "model" program:

```
subroutine head(x,y)
  double precision,intent(in) :: x
  double precision,intent(out) :: y
!$openad INDEPENDENT(x)
  y=sin(x*x)
!$openad DEPENDENT(y)
end subroutine
```

preaccumulation & store $\boldsymbol{J}_i$:

```
...
oadS_0 = (X%v*X%v)
Y%v = SIN(oadS_0)
oadS_2 = X%v
oadS_3 = X%v
oadS_1 = COS(oadS_0)
oadS_4 = (oadS_2 * oadS_1)
oadS_5 = (oadS_3 * oadS_1)
oadD(oadD_ptr) = oadS_4
oadD_ptr = oadD_ptr+1
oadD(oadD_ptr) = oadS_5
oadD_ptr = oadD_ptr+1
...
```

retrieve stored $\boldsymbol{J}_i$ & propagate:

```
...
oadD_ptr = oadD_ptr-1
oadS_6 = oadD(oadD_ptr)
X%d = X%d+Y%d*oadS_6
oadD_ptr = oadD_ptr-1
oadS_7 = oadD(oadD_ptr)
X%d = X%d+Y%d*oadS_7
Y%d = 0.0d0
...
```

sidebar: toy example - reverse mode

same code preparation ⇒ reverse mode OpenAD pipeline

numerical "model" program:

```
subroutine head(x,y)
  double precision,intent(in) :: x
  double precision,intent(out) :: y
!$openad INDEPENDENT(x)
  y=sin(x*x)
!$openad DEPENDENT(y)
end subroutine
```

preaccumulation & store J_i:

```
...
oadS_0 = (X%v*X%v)
Y%v = SIN(oadS_0)
oadS_2 = X%v
oadS_3 = X%v
oadS_1 = COS(oadS_0)
oadS_4 = (oadS_2 * oadS_1)
oadS_5 = (oadS_3 * oadS_1)
oadD(oadD_ptr) = oadS_4
oadD_ptr = oadD_ptr+1
oadD(oadD_ptr) = oadS_5
oadD_ptr = oadD_ptr+1
...
```

retrieve stored J_i & propagate:

```
...
oadD_ptr = oadD_ptr-1
oadS_6 = oadD(oadD_ptr)
X%d = X%d+Y%d*oadS_6
oadD_ptr = oadD_ptr-1
oadS_7 = oadD(oadD_ptr)
X%d = X%d+Y%d*oadS_7
Y%d = 0.0d0
...
```

sidebar: toy example - reverse mode

same code preparation ⇒ reverse mode OpenAD pipeline
⇒ adapt the driver routine

numerical “model” program:

```
subroutine head(x,y)
  double precision,intent(in) :: x
  double precision,intent(out) :: y
!$openad INDEPENDENT(x)
  y=sin(x*x)
!$openad DEPENDENT(y)
end subroutine
```

driver modified for reverse mode:

```
program driver
  use OAD_active
  implicit none
  external head
  type(active):: x, y
  x%v=.5D0
  y%d=1.0
  our_rev_mode%tape=.TRUE.
  call head(x,y)
  print *, "F(1,1)=",x%d
end program driver
```

preaccumulation & store J_i:

```
...
oadS_0 = (X%v*X%v)
Y%v = SIN(oadS_0)
oadS_2 = X%v
oadS_3 = X%v
oadS_1 = COS(oadS_0)
oadS_4 = (oadS_2 * oadS_1)
oadS_5 = (oadS_3 * oadS_1)
oadD(oadD_ptr) = oadS_4
oadD_ptr = oadD_ptr+1
oadD(oadD_ptr) = oadS_5
oadD_ptr = oadD_ptr+1
...
```

retrieve stored J_i & propagate:

```
...
oadD_ptr = oadD_ptr-1
oadS_6 = oadD(oadD_ptr)
X%d = X%d+Y%d*oadS_6
oadD_ptr = oadD_ptr-1
oadS_7 = oadD(oadD_ptr)
X%d = X%d+Y%d*oadS_7
Y%d = 0.0d0
...
```

sidebar: toy example - reverse mode

same code preparation ⇒ reverse mode OpenAD pipeline
⇒ adapt the driver routine

numerical "model" program:

```
subroutine head(x,y)
  double precision,intent(in) :: x
  double precision,intent(out) :: y
!$openad INDEPENDENT(x)
  y=sin(x*x)
!$openad DEPENDENT(y)
end subroutine
```

driver modified for reverse mode:

```
program driver
  use OAD_active
  implicit none
  external head
  type(active):: x, y
  x%v=.5D0
  y%d=1.0
  our_rev_mode%tape=.TRUE.
  call head(x,y)
  print *, "F(1,1)=",x%d
end program driver
```

preaccumulation & store J_i:

```
...
oadS_0 = (X%v*X%v)
Y%v = SIN(oadS_0)
oadS_2 = X%v
oadS_3 = X%v
oadS_1 = COS(oadS_0)
oadS_4 = (oadS_2 * oadS_1)
oadS_5 = (oadS_3 * oadS_1)
oadD(oadD_ptr) = oadS_4
oadD_ptr = oadD_ptr+1
oadD(oadD_ptr) = oadS_5
oadD_ptr = oadD_ptr+1
...
```

retrieve stored J_i & propagate:

```
...
oadD_ptr = oadD_ptr-1
oadS_6 = oadD(oadD_ptr)
X%d = X%d+Y%d*oadS_6
oadD_ptr = oadD_ptr-1
oadS_7 = oadD(oadD_ptr)
X%d = X%d+Y%d*oadS_7
Y%d = 0.0d0
...
```

try it out

- `cd Examples/OneMinuteReverse`
- note identical `head.prepped.f90` but changed flag in the `Makefile`
- run `make clean` and `make`
- should produce output like this:

```
openad -c -m rj head.prepped.f90
openad log: openad.2009-09-08_00:44:10.log~
parsing head.prepped.f90
analyzing source code and translating to xaif
adjoint transformation
 getting runtime support file OAD_active.f90
 getting runtime support file w2f__types.f90
 getting runtime support file iaddr.c
 getting runtime support file ad_inline.f
 getting runtime support file OAD_cp.f90
 getting runtime support file OAD_rev.f90
 getting runtime support file OAD_tape.f90
 getting template file
translating transformed xaif to whirl
unparsing transformed whirl to fortran
postprocessing transformed fortran
gfortran -o OAD_active.o -c OAD_active.f90
gfortran -o OAD_cp.o -c OAD_cp.f90
gfortran -o OAD_tape.o -c OAD_tape.f90
gfortran -o OAD_rev.o -c OAD_rev.f90
cc -o iaddr.o -c iaddr.c
gfortran -o head.prepped.xb.x2w.w2f.pp.o -c head.prepped.xb.x2w.w2f.pp.f
gfortran -o driver w2f__types.o OAD_active.o OAD_cp.o OAD_tape.o OAD_rev.o
   iaddr.o driver.o head.prepped.xb.x2w.w2f.pp.o
```

try it out contd. ...

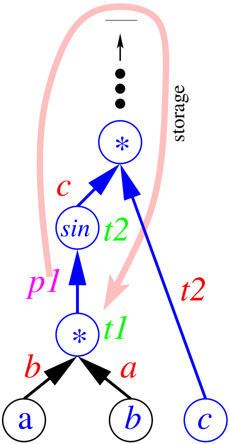
- look at the transformed file `head.prepped.xb.x2w.w2f.pp.f`
- look at the the driver code in `driver.f90`
- run the binary: `./driver`
- should produce output like this:

```
/home/guest1/OpenAD/Examples/OneMinuteReverse> ./driver
 driver running for x =  0.50000000000000000
            yields y =  0.54630248984379048        dy/dx =   1.2984464104095248
     1+tan(x)^2-dy/dx =  3.85975973404839579E-017
```

Reversal Schemes

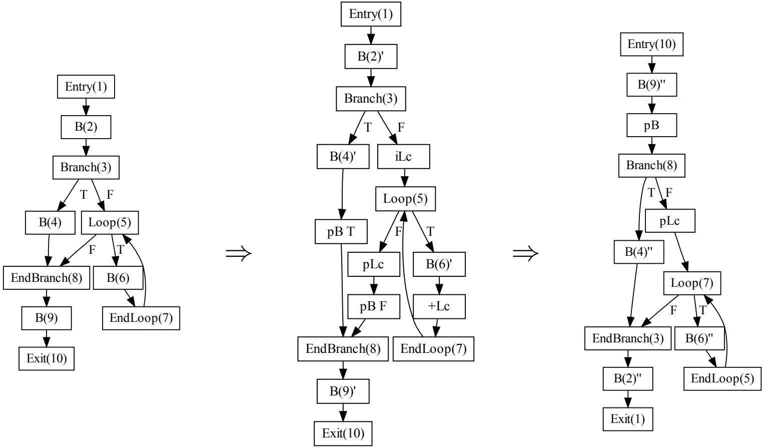
- why it is needed
- major modes
- OpenAD implementation
- alternatives

recap: store intermediate values / partials



storage also needed for control flow trace and addresses...

original CFG ⇒ record a path through the CFG ⇒ adjoint CFG

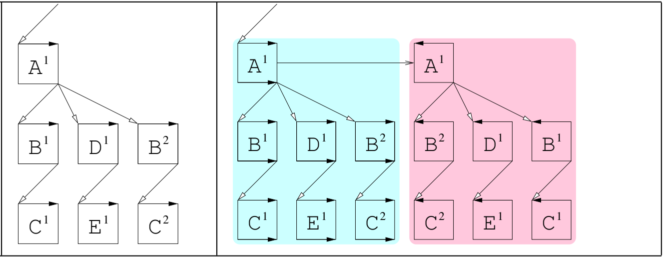


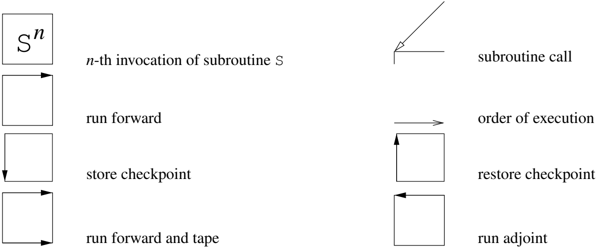
often cheap with structured control flow and simple address computations (e.g. index from loop variables)

unstructured control flow and pointers are expensive

trace all at once = global *split* mode

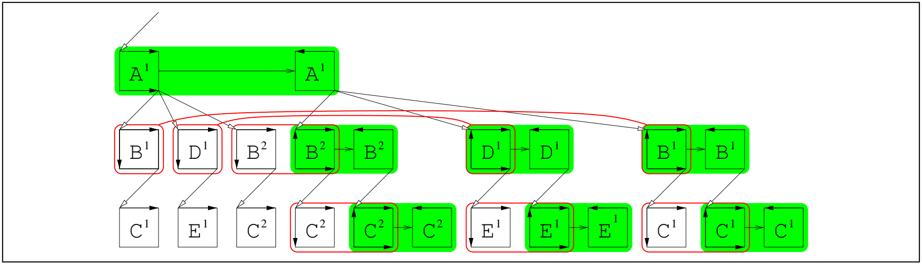
```
subroutine A()
  call B(); call
D(); call B();
end subroutine A
subroutine B()
  call C()
end subroutine B
subroutine C()
  call E()
end subroutine C
```





- have memory limits - need to create tapes for short sections in reverse order
- subroutine is “natural” checkpoint granularity, different mode...

trace one SR at a time = global *joint* mode



taping-adjoint pairs

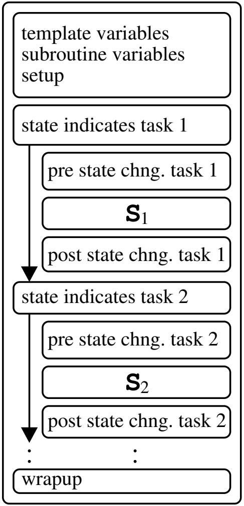
checkpoint-recompute pairs

the deeper the call stack - the more recomputations (unimplemented solution - result checkpointing)

familiar tradeoff between storing and recomputation at a higher level but in theory can be all unified.

in practice - hybrid approaches...

in OpenAD orchestrated with templates




```
subroutine template()
  use OAD_tape ! tape storage
  use OAD_rev  ! state structure
!$TEMPLATE_PRAGMA_DECLARATIONS
  if (rev_modetape) then
! the state component
! 'taping' is true
!$PLACEHOLDER_PRAGMA$ id=2
  end if

  if (rev_modeadjoint) then
! the state component
! 'adjoint' run is true
!$PLACEHOLDER_PRAGMA$ id=3
  end if

end subroutine template
```

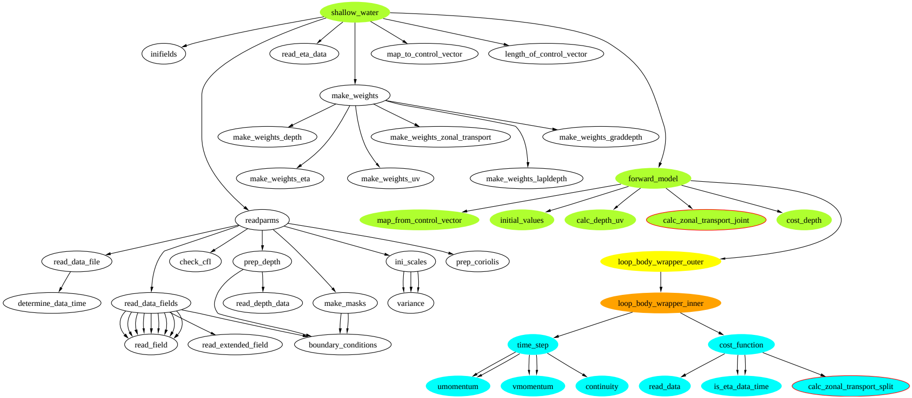
ingredients

- OpenAnalysis has *side-effect analysis*
- provides checkpoint sets as (formal) arguments & references to global variables
- we ask for four sets: `ModLocal` $\subseteq$ `Mod`, `ReadLocal` $\subseteq$ `Read`

look at some code:

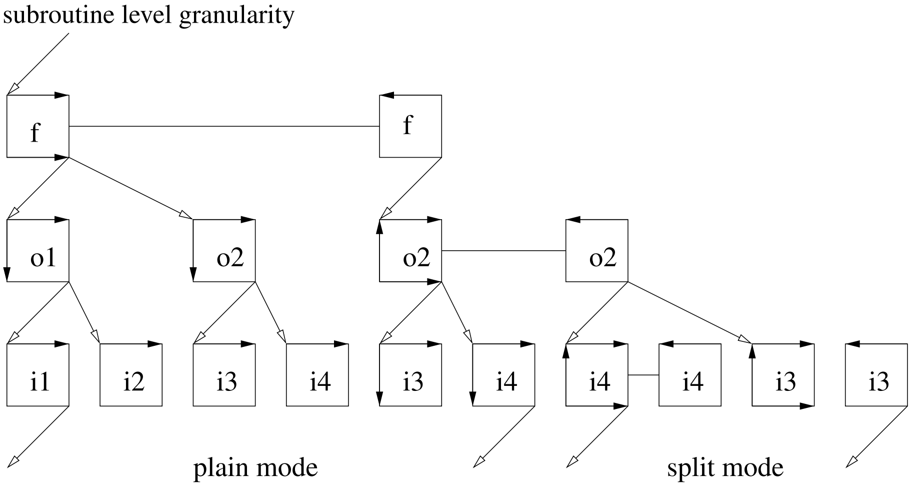
- a simple split mode template in `OpenAD/runTimeSupport/simple/ad_template.split.f`
- look at the joint mode template file in `OpenAD/Examples/OneMinuteReverse`
- change the driver and rerun the example with `-m rs` instead of `-m rj`
- look at the output
- a bit more complicated - the ShallowWater example

example - call graph of a shallow water model



- mix joint and split mode
- nested loop checkpointing in outer and inner loop body wrapper
- inner loop body in split mode
- `calc_zonal_transport` is used in both contexts

reversal scheme with nested checkpointing



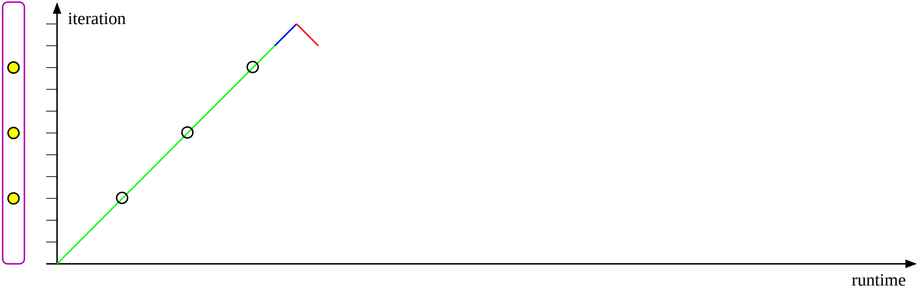
use of checkpointing to mitigate storage requirements

- 11 iters.

use of checkpointing to mitigate storage requirements

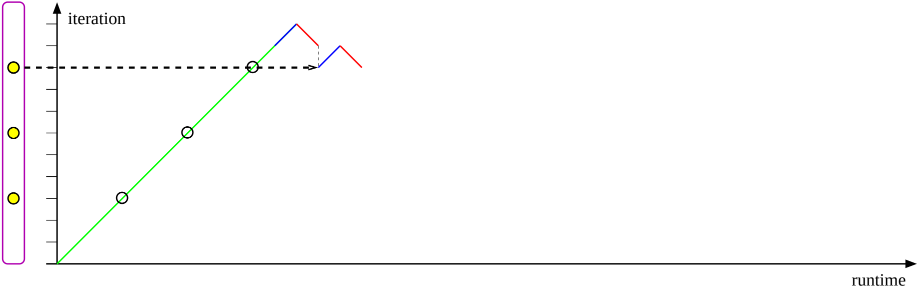
- 11 iters., memory limited to one iter. of storing $\boldsymbol{J}_i$
- run forward, store the last step, and adjoin

use of checkpointing to mitigate storage requirements



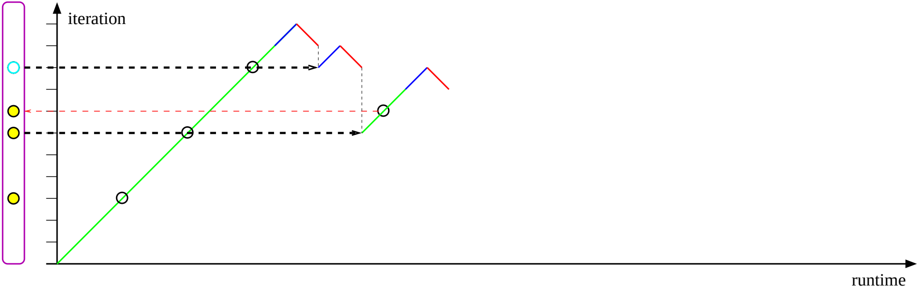
- 11 iters., memory limited to one iter. of storing $\boldsymbol{J}_i$ & $\boxed{\text{3 checkpoints}}$
- run forward, store the last step, and adjoin

use of checkpointing to mitigate storage requirements



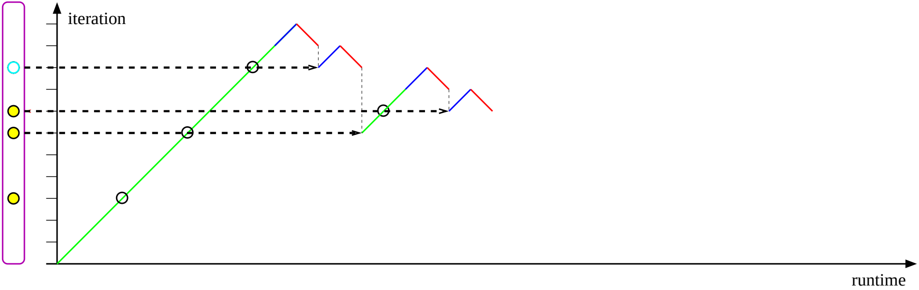
- 11 iters., memory limited to one iter. of storing $\boldsymbol{J}_i$ & $\boxed{\text{3 checkpoints}}$
- run forward, store the last step, and adjoin
- restore checkpoints and recompute

use of checkpointing to mitigate storage requirements



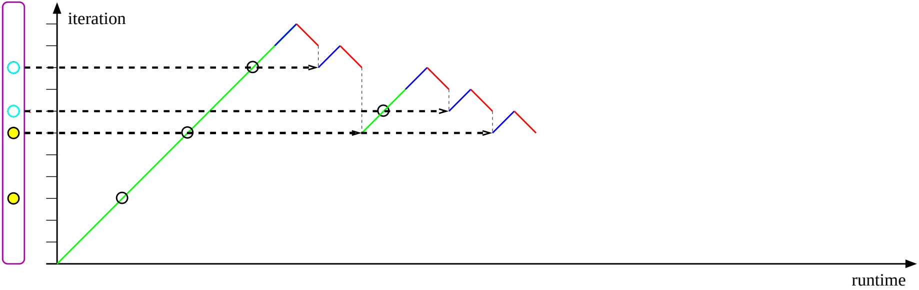
- 11 iters., memory limited to one iter. of storing J_i & $\boxed{\text{3 checkpoints}}$
- run forward, store the last step, and adjoin
- restore checkpoints and recompute (2 levels in this example)
- reuse checkpoint space as it becomes available for new checkpoints

use of checkpointing to mitigate storage requirements



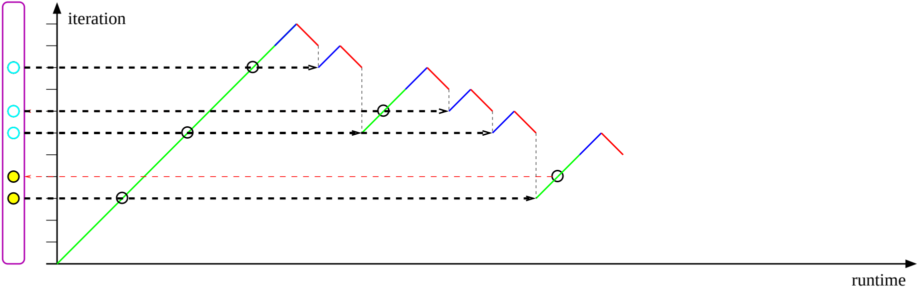
- 11 iters., memory limited to one iter. of storing J_i & $\boxed{\text{3 checkpoints}}$
- run forward, store the last step, and adjoin
- restore checkpoints and recompute (2 levels in this example)
- reuse checkpoint space as it becomes available for new checkpoints

use of checkpointing to mitigate storage requirements



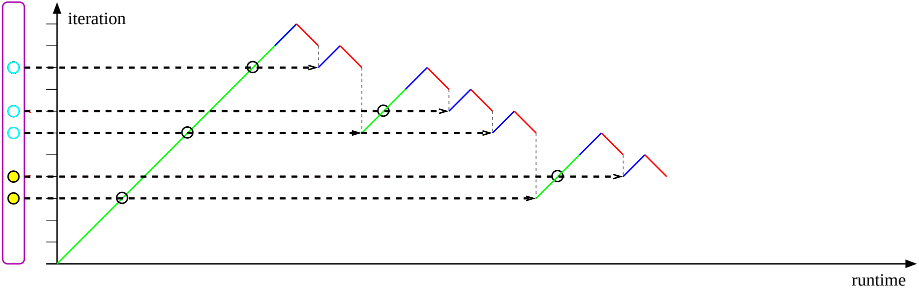
- 11 iters., memory limited to one iter. of storing J_i & $\boxed{\text{3 checkpoints}}$
- run forward, store the last step, and adjoin
- restore checkpoints and recompute (2 levels in this example)
- reuse checkpoint space as it becomes available for new checkpoints

use of checkpointing to mitigate storage requirements



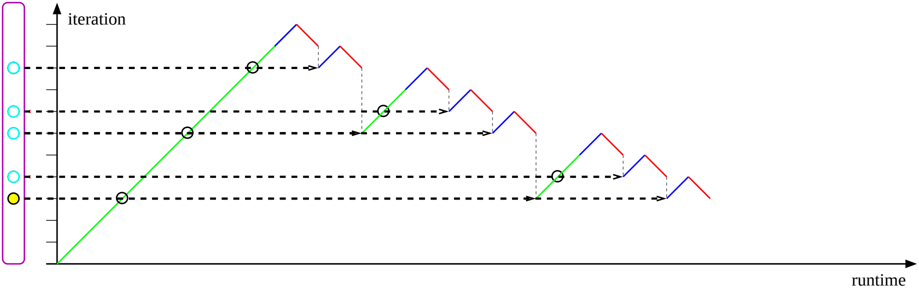
- 11 iters., memory limited to one iter. of storing J_i & $\boxed{\text{3 checkpoints}}$
- run forward, store the last step, and adjoin
- restore checkpoints and recompute (2 levels in this example)
- reuse checkpoint space as it becomes available for new checkpoints

use of checkpointing to mitigate storage requirements



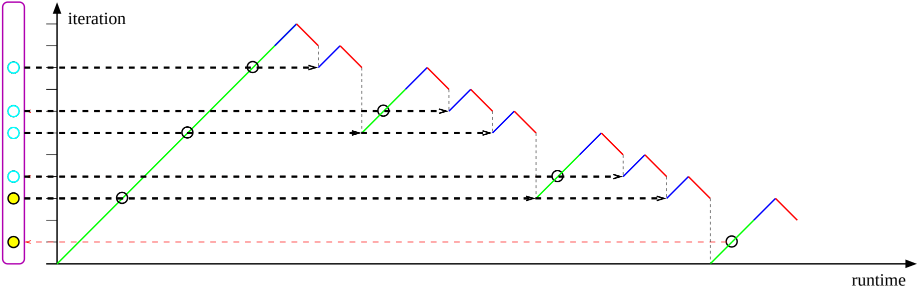
- 11 iters., memory limited to one iter. of storing J_i & $\boxed{\text{3 checkpoints}}$
- run forward, store the last step, and adjoin
- restore checkpoints and recompute (2 levels in this example)
- reuse checkpoint space as it becomes available for new checkpoints

use of checkpointing to mitigate storage requirements



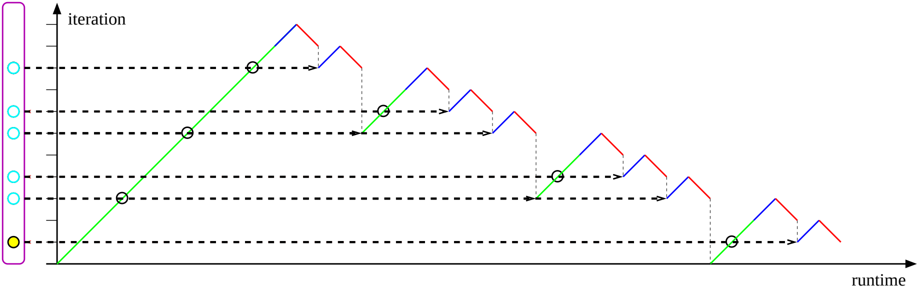
- 11 iters., memory limited to one iter. of storing $\boldsymbol{J}_i$ & $\boxed{\text{3 checkpoints}}$
- run forward, store the last step, and adjoin
- restore checkpoints and recompute (2 levels in this example)
- reuse checkpoint space as it becomes available for new checkpoints

use of checkpointing to mitigate storage requirements



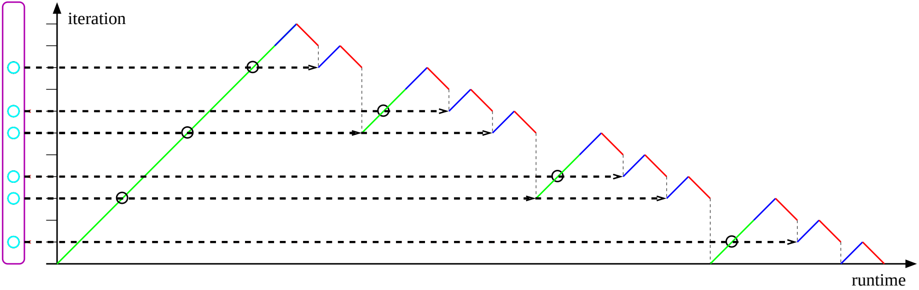
- 11 iters., memory limited to one iter. of storing $\boldsymbol{J}_i$ & $\boxed{\text{3 checkpoints}}$
- run forward, store the last step, and adjoin
- restore checkpoints and recompute (2 levels in this example)
- reuse checkpoint space as it becomes available for new checkpoints

use of checkpointing to mitigate storage requirements



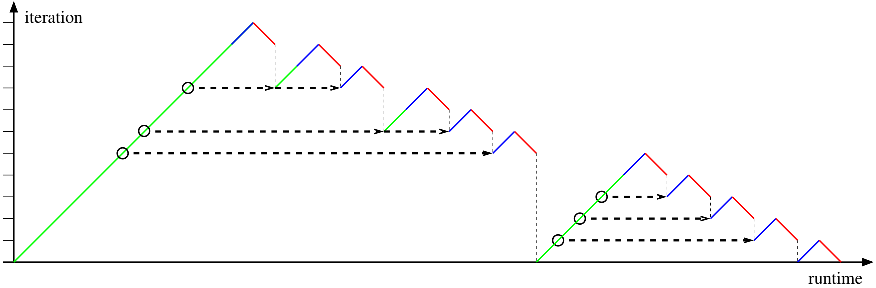
- 11 iters., memory limited to one iter. of storing $\boldsymbol{J}_i$ & $\boxed{\text{3 checkpoints}}$
- run forward, store the last step, and adjoin
- restore checkpoints and recompute (2 levels in this example)
- reuse checkpoint space as it becomes available for new checkpoints

use of checkpointing to mitigate storage requirements



- 11 iters., memory limited to one iter. of storing $\boldsymbol{J}_i$ & $\boxed{\text{3 checkpoints}}$
- run forward, store the last step, and adjoin
- restore checkpoints and recompute (2 levels in this example)
- reuse checkpoint space as it becomes available for new checkpoints

use of checkpointing to mitigate storage requirements



- 11 iters., memory limited to one iter. of storing J_i & $\boxed{\text{3 checkpoints}}$
- run forward, store the last step, and adjoin
- restore checkpoints and recompute (2 levels in this example)
- reuse checkpoint space as it becomes available for new checkpoints
- optimal (binomial) scheme encoded in `revolve`; F9X implementation available at `http://mercurial.mcs.anl.gov/ad/RevolveF9X` (tomorrow)

usage concerns

Adjoint efficiency depends on AD transformation algorithms and exploiting higher level model properties (sparsity, iterative solvers, self adjointness,...)

BUT source transformation efficiency depends also on

- **capability for structured control flow reversal**
- **code analysis accuracy**
- **partitioning the execution for checkpointing**

the above are affected by

- use of programming language features
- using such features in certain inherently difficult to handle patterns
- programming style

therefore

- knowing some AD tool “internal” algorithms is of interest to the user (e.g. compare to compiler vectorization or interval arithmetic)
- only very simple models with low computational complexity $\rightarrow$ can get away with “something”
- fully automatic solutions exist for narrowly defined setups (e.g. NEOS)

When dealing with any unsupported language feature / programming pattern :

- Does it have a supported alternative and is the alternative more efficient (and better maintainable in the model source)?
- Is the adjoint of such an alternative more efficient than the adjoint of the unsupported construct?
- What is the effort of changing the model vs. the effort of implementing a potentially complicated or rarely used or inherently inefficient adjoint transformation?

OpenAD mode of operation: implement language features on demand so that we can maximize the time available to improve the generally applicable AD algorithms!

Separating the numerical core

want precise compile-time data flow analysis (activity, side effect, etc...)

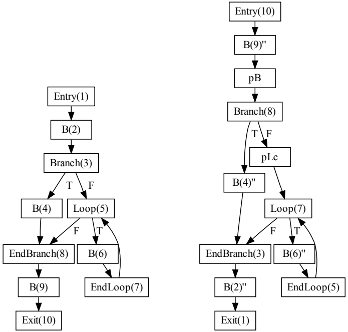
have conservative overestimate of aliasing, MOD sets, ...

to reduce the overestimate:

- encapsulate ancillary logic (monitoring, debugging, timing, I/O,...)
- small modules, routines, source files (good coding practice anyway)
- consider separate modules for data and interfaces
- extraction via source file selection
- filtered-out routines treated as "black box", with optimistic(!) assumptions
- provide stubs when optimistic assumptions are inappropriate
- transformation shielded from dealing with non-numeric language features
- note: the top level model *driver* needs to be manually adjusted

Structured vs. Unstructured Control Flow

- think - `GOTO`, alternative `ENTRY`, early `RETURN`, ...
- structured control flow is characterizable by some control flow graph properties; permits **structured reverse control flow**!
- simple view: **use only loops and branches** and no other control flow constructs (some things are easily fixable though, e.g. turn `STOP`s into some error routine call ,...)
- example: early return from within a loop (CFG left, adjoint CFG right)

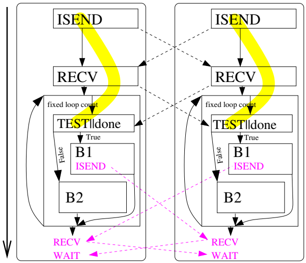


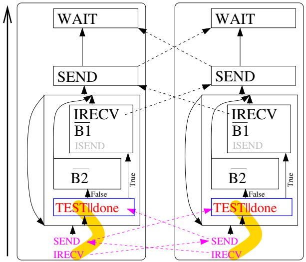
- all is fine without the red arrow
- by inspection: adjoint needs alternative `ENTRY` (or `GOTO`); but difficult to automate in general
- need to trace more control flow path details
- unstructured control flow is bad for compiler optimization, already for the original model!
- possible generic but inefficient fallback: trace enumerated basic blocks, replay inverse trace with `GOTO <blockId>` (no branches/loops left, more memory needed for trace)

Non-deterministic control flow

= control flow may change between two model executions on identical model inputs because of a multiuser system environment
examples:

- branching based on availability of system resources (that may be used by others), disk space, memory, system load
- communication in parallel execution for instance with mutexes, semaphores, (justified) use of MPI_TEST (test for completion of one exchg. 1 to early start exhg. 2, adjoint needs to switch test to exchg.2)





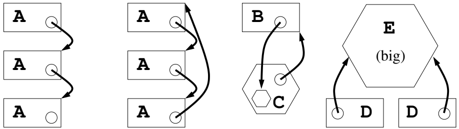
Non-deterministic control flow II

- hard to **automatically** detect the context to which a tested condition applies but the transformation requires the context information to correctly generate & place the adjoint test condition
- non-deterministic communication with MPI wildcards can be made deterministic (at the expense of lower efficiency) by recording the actual wild card values and using them in the adjoint sweep.
- google “adjoinable MPI”

Checkpointing and non-contiguous data

checkpointing = saving program data (to disk)

- "contiguous" data: scalars, arrays (even with stride > 1), strings, structures,...
- "non-contiguous" data: linked lists, rings, structures with pointers,...
- checkpointing is very similar to "serialization"
- Problem: decide when to follow a pointer and save what we point to



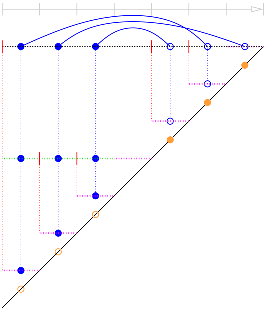
- unless we have extra info this is not decidable at source transformation time
- possible fallback: runtime bookkeeping of things that have been saved (is computationally expensive)

Semantically Ambiguous Data

- e.g. `EQUIVALENCE` (or its C counterpart `union`)
  - data dependence analysis: dependencies propagate from one variable to **all** equivalenced variables
  - "activity" (i.e. the need to generate adjoint code for a variable) leaks to all equivalenced variables whether appropriate or not
  - certain technical problems with the use of an active type (as in OpenAD)
- work-arrays (multiple,0 semantically different fields are put into a (large) work-array); access via index offsets
  - data dependence analysis: there is *array section analysis* but in practice it is often not good enough to reflect the implied semantics
  - the entire work-array may become active / checkpointed
- programming patterns where the analysis has no good way to track the data dependencies:
  - data transfer via files (don't really want to assume all read data depends on all written data)
  - non-structured interfaces: exchanging data that is identified by a "string" as done for instance in the ESMF interfaces (if you feel bad about Fortran think of `void*` in C.)

Recomputation from Checkpoints and Program Resources

think of memory, file handles, sockets, MPI communicators,...



- problem when resource allocation and deallocation happen in different partitions (see hierarchical checkpointing scheme in the figure on the left)
- current AD checkpointing **does not track resources**
- dynamic memory is “easy” as long as nothing is deallocated before the adjoint sweep is complete.

options to handle local deallocations

```
1  subroutine foo(p,t)
2  integer, intent(inout), pointer, dimension(:) :: p
3  integer, target :: t(:)
4  t=2*p ! need adjoint pointer to point to (invisible) t1
5  p=>t ! pointer is overwritten
6  end subroutine
7
8  subroutine bar
9  interface
10 subroutine foo(p,t)
11 integer, intent(inout), pointer, dimension(:) :: p
12 integer, target :: t(:)
13 end subroutine
14 end interface
15 integer, target, allocatable :: t1(:), t2(:)
16 integer, pointer, dimension(:) :: p
17 allocate(t1(1)); allocate(t2(1))
18 t1(1)=1
19 p=>t1
20 call foo(p,t2)
21 print*, p(1) ! p points now to t2
22 end subroutine ! t1 and t2 are deallocated
23
24 program p
25 call bar()
26 end program
```

- modify model to reuse/grow allocated memory (rather than repeatedly allocate/deallocate), e.g. turn `t1 t2` into global vars,...
- potential solution for allocate/deallocate **within a checkpointing partition without pointers**: track allocated memory to turn deallocates (here implicit on exit line 22) into allocates (of the appropriate size)
- potential (complicated) solution when pointers are involved: associate dynamic allocations in forward sweep to dynamic allocations in the adjoint sweep (adjoint needs to restore pointer overwritten on line 5, but stored pointer *value* references deallocated memory; need abstract association between forward allocate on line 17 and adjoint allocate corresponding to implicit deallocate on line 22)

summary so far

overview of AD concepts and some OpenAD examples

motivation for the following recommendations:

summary so far

overview of AD concepts and some OpenAD examples

motivation for the following recommendations:

- separation of the numerical core

summary so far

overview of AD concepts and some OpenAD examples

motivation for the following recommendations:

- separation of the numerical core
- unambiguous data and interfaces

summary so far

overview of AD concepts and some OpenAD examples

motivation for the following recommendations:

- separation of the numerical core
- unambiguous data and interfaces
- well structured code

summary so far

overview of AD concepts and some OpenAD examples

motivation for the following recommendations:

- separation of the numerical core
- unambiguous data and interfaces
- well structured code
- allocate once

summary so far

overview of AD concepts and some OpenAD examples

motivation for the following recommendations:

- separation of the numerical core
- unambiguous data and interfaces
- well structured code
- allocate once
- avoid gratuitous use of pointers

summary so far

overview of AD concepts and some OpenAD examples

motivation for the following recommendations:

- separation of the numerical core
- unambiguous data and interfaces
- well structured code
- allocate once
- avoid gratuitous use of pointers
- model development with AD in mind $\sim$ good coding practice anyway